Producing a Global Requirement Model from Multiple Requirement Specifications

Erwan Brottier, Benoit Baudry, Yves Le Traon, David Touzet, Bertrand Nicolas

Producing a Global Requirements Model from Multiple Requirement Specifications

Erwan Brottier[1], Benoit Baudry[2], Yves Le Traon[3], David Touzet[2], Bertrand Nicolas[1]
[1] France Télécom R&D, 2 av. Pierre Marzin 22307 Lannion Cedex, France
{erwan.brottier, bertrand.nicolas}@orange-ftgroup.com
[2] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
{bbaudry, dtouzet}@irisa.fr
[3] ENST Bretagne, 2 rue de la Châtaigneraie, CS 17607, 35576 Cesson Sévigné Cedex
yves.letraon@enst-bretagne.fr

Abstract. *Requirements documentation is a collection of partial specifications produced by different stakeholders. Obtaining a global specification is a fundamental step of a requirement analysis process. Merging requirement specifications is indeed a way to reveal inconsistencies between them. We propose in this paper a model-driven mechanism for that purpose. It takes as inputs a set of texts or models which conform to input requirement languages and produces a global requirements model. This mechanism is integrated in a platform called R2A which stands for "requirements to analysis". The R2A core element is its core requirement metamodel which has been defined for capturing the global requirements model. We illustrate our approach with requirement specifications expressed in a constrained natural language. This platform and its mechanism have been completely implemented with MDE* (Model Driven Engineering) *technologies. As such, it is a good example of how MDE technologies can contribute to requirements engineering as a technical solution.*

1 Introduction

Requirements Engineering (RE) is the first step of a software development process. It consists in discussing, modeling, validating and agreeing on the requirements. It is critical since the primary measure of software system success is the degree to which it meets the purpose for which it was intended [1]. It is now well-known that the first reason of software project failures is requirement issues such as misunderstanding of the actual needs, requirement inconsistencies or even lack of requirements documentation [2, 3]. Performing a good RE process has a strong impact on the relevance of a software development process and consequently the quality of the system itself.

Requirements capture a wide range of concerns, expressed by a significant number of stakeholders (end users, domain experts, project leaders, business analysts, software and requirement engineers, etc.). Stakeholders produce a set of specifications according to their viewpoints and expressed using different syntaxes adapted to their domain.

The requirement distribution in heterogeneous documents revealing several partial specifications is a good solution to let all stakeholders express their needs. However, it makes difficult the analysis of the software specification since semantic links between partial specifications remain implicit. Indeed, partial specifications are likely to be semantically interlinked since parts of them can describe the same aspect of the software specification. A robust mechanism is needed for gathering all stakeholders' specifications in a single model in order to validate semantics, completeness and consistency of these viewpoints. This model must capture the global problem space for the system.

Current RE tools allow either a single input global requirement specification or single requirement syntax. In both cases, they are not designed to evolve easily if the input language must be customized to stakeholders' preferences and the kind of software being developed. That is the main reason why these tools are not widely used in an industrial scale.

This paper presents a general model-driven process (and tools) to produce a global requirements model from a set of specifications written in different syntaxes. It does not aim to detect or resolve inconsistencies which may appear in the resulting model. The process is composed of two steps. The first one, required for textual languages, parses each textual requirement specification to produce an abstract syntax model. The second step (i) interprets the semantics of each abstract syntax model in an intermediate requirements model, (ii) merges these models into a global requirements model. These two steps (and included sub-steps) use MDE (Model Driven Engineering) techniques. This process produces a global requirements model on which it is possible to detect underspecified elements or inconsistencies between different specifications. However, the analysis of the global model is out of the scope of this paper.

In the paper, we illustrate this process with the R2A (*"requirements to analysis")* platform, which provides MDE mechanisms for interpreting, analyzing and simulating requirements. Its core asset is a metamodel that defines the modeling language we use to capture requirements. It captures both business domain and functional concerns of system requirements as use cases guarded with specific business rules represented as pre- and post-conditions. The main R2A input textual language is a constrained natural language called the "Requirement Description Language" (RDL). The result of merging RDL specifications is a model which conforms to the R2A metamodel.

This paper is organized as follows. Section 2 discusses the need of a merge mechanism and outlines by an example its application scope. Section 3 presents the general merge mechanism, and introduces its implementation in the R2A platform. The two following sections detail the relevant parts of the platform that implement this mechanism. Section 4 introduces RDL and the R2A metamodel; section 5 presents the interpretation and fusion transformations involved in the R2A import step. Finally, we discuss the related works in section 6 and conclude in section 7.

2 Merging as a Basis for Requirements Engineering Activities

2.1 Managing a Set of Requirements Specifications

RE is the part of the software engineering which focuses on development issues at the early stage of a development process. Requirements engineers have to deal with requirements produced by all stakeholders, in order to gather sufficient and consistent information about what the software has to be.

> **Definition – Requirement and requirement specification***: A requirement is a constraint on the software-to-be. A requirement specification is a set of requirements which describe what the software has to resolve (in other terms a problem) and why the software is required (the goals).*

Requirements are specified by heterogeneous stakeholders such as end users, domain experts, project leaders, business analysts, software and requirement engineers and so forth. Moreover, stakeholders focus on different aspects of the software and have different needs, expertise and skills. The result of discussions between stakeholders is a collection of partial specifications, also called *viewpoints* [4]. We provide below a definition of partial specification:

> **Definition - Partial requirement specification or Viewpoint***: A partial requirement specification produced by one particular stakeholder or stakeholders group. It is an incomplete viewpoint on the software-to-be.*

Partial specifications describe different points of view on the same object that is the software-to-be. As such they are likely to be semantically interlinked [4]. They must be merged into only one model that reflects the global complexity of the requirements. By this way, implicit semantic links can be expressed so that automatic treatments can be expressed. The result of this process is the global requirement specification.

This merge process is a basis for RE activities. It produces a global requirements model which is necessary for computing automatic treatments. It is a way for analyzing requirements and discovering inconsistencies between partial specifications. It can reveal communication issues between stakeholders.

2.2 An Example

Figure 1 gives excerpts from a "*Library Management System*" called *LMS*. (a) and (b) textual partial specifications have been produced separately. The first one has been written by the library end users while the second corresponds to the librarian's viewpoint. These two partial specifications must be merged to obtain a more global specification.

These requirements capture functional concerns. They define basic actions provided by the LMS which are an interface between the system and its environment. Each sentence defines one action property which is either an activation condition or an effect if triggered. For instance, the sentences (1, 6, 7, 8) describe activation conditions for *borrow* action while the sentence (2) defines one effect: the given *book* is now borrowed by the *customer* who triggers the action.

Business domain entities are also described by these partial requirements. It is possible to deduce from them a dictionary of business concepts for the LMS (*book*, *customer* and *librarian*). The two last business concepts could be identify as actors since they are subjects of action verbs. The sentence (2) suggests that *book* and *customer* concepts are linked by a dependence called *borrowed*. The sentence (5) implies that a *book* can be in a state called *damaged*, and so on.

The LMS will be used through the entire paper to illustrate our approach. Its global requirements are described below:

- A library is maintained by a librarian.
- A customer must register in the library to avail the facility of borrowing the books.

(1) The "book" must be not “borrowed” before the "customer" can "borrow" the "book".
(2) The "book" is "borrowed" by the customer after the "customer" did "borrow" the "book".
(3) The “book” must be “borrowed” by the customer before the customer can “return” the “book”.
(4) The "book" becomes not "borrowed" after the "customer" did "return" the "book".
(5) The "book" is "damaged" after the "customer" did "damage" the "book". (a)

(6) The "customer" must be "registered" before the "customer" can "borrow" the "book".
(7) The "book" must be not "damaged" before the "customer" can "borrow" the "book".
(8) The "book" must be “registered” before the "customer" can "borrow" the "book".
(9) The "customer" becomes "registered" after the "librarian" did "subscribe" the "customer".
(10) The "book" becomes "registered" after the "librarian" did "register" the "book".
(11) The "book" is not “borrowed” after the "librarian" did "register" the "book".
(12) The "book" is not "registered" after the "librarian" did "unregister" the "book".
(13) Each "book" is not "damaged" after the "librarian" did "make an inventory". (b)

Figure 1 - A piece of textual requirement specifications of a library management system

- Books must be registered before being available to the customers.
- A Customer can borrow and return books.
- When a customer returns a damaged book, the book is not available for customers, till the librarian performs an inventory check.

2.3 Characterization of a Merge Operation

Merging two partial specifications consists of making explicit the semantic links between them. By observing the previous requirements, two distinct types of semantic links can be identified:

- *Equivalence link (or explicit link)*: it means that linked elements represent the same concept. For instance, while the concept of *book* is unique, the expression “book” surrounded by quote is present 22 times in Figure 1. This redundancy is not a problem if the text is only used as documentation. It is even natural in textual representations. But each reference to a same concept must be represented by only one element in a model dedicated to analysis purpose.

  However, redundancy is not always a mark of equivalence link. The two words “registered” in requirements (8) and (9) do not refer the same meaning. The first one is a state of a *book* while the other one is a state of a *customer*. Then, an equivalence link can not be systematically deduced from words equality. It is defined according to the overall specification.

  Two elements can represent the same concept even if their types are different. For instance, the *customer* is an actor since it appears as one subject in requirement (4) while it is considered as a business entity in the requirement (9).
- *Implicit link*: it is a semantic link between elements. For instance, the guards of the *borrow* action are defined by requirements (1, 6, 7, 8). Semantically, these conditions are logically linked by conjunctions since all of them must be true for triggering the action. These conjunction links must appear in the resulting global requirements model while they are implicit in Figure 1.

3 A Model-Driven Process for Producing a Global Requirements Model

Merging partial requirement specifications is crucial during a RE process. The concerns that requirements capture are tightly related to the business domain of the software. Companies have their own requirement language, according to their cultures and habits. So a merge process must provide flexibility to support different input languages. Moreover, the global requirements model issued from the merge operation must be easy to manipulate. This section proposes a general process which satisfies these requirements.

3.1 The Model-Driven Engineering (MDE) as a Technical Solution

Model-Driven Engineering (MDE) [5] advocates the systematic use of models for designing and implementing software. The most famous MDE initiative is the MDA (Model Driven Architecture) [6], which has been defined by the OMG for making software system implementations independent to a particular execution platform.

MDE technologies are mainly based on two notions: metamodel and model transformation. A model is structured data which conform to a metamodel in the same way as a source program conforms to a grammar. Model transformations are specific programs which are dedicated to model manipulation.

Requirements are the first models of the software-to-be. Requirement analysis activities consist essentially of the manipulation of these models. Model-driven technologies are thus particularly adapted to implement requirement

analysis tools. We propose a MDE process for manipulating a set of partial textual requirements to produce a global requirements model.

3.2 A Two Steps Process for Merging Partial Requirement Specifications

The overall process is completely based on a MDE approach. It is build around a core element which is the global requirements model. Its metamodel describes which requirement concerns can be captured. In a requirement analysis context, this metamodel can be for instance the input language of such a formal analysis tool or the conceptual modeling language of an elicitation process. Input languages are chosen so that the union of their expressiveness is sufficient w.r.t. the global requirement metamodel.

Figure 2 depicts an overview of the MDE process with its models, metamodels and transformations. The process is decomposed in two sequential steps. The result of applying these steps on one partial textual specification is the addition of its semantics in the global requirements model. These steps are:

1 - **Parsing**. This step brings partial requirement textual specifications into the model world. It produces a *partial requirement specification model* equivalent to the *textual specification*. The parsing step is specific to the input requirement language.

2 – **Import**. This step is performed by a model transformation. It aims to import the information of the partial requirement specification models into the global requirements model. It is composed of two sub-steps. The first one, called *interpretation*, expresses specification models in terms of the core requirement metamodel. Its result for one partial requirement specification model is one *intermediate model*. An intermediate model is a set of model fragments which are instances of the core requirement metamodel. The second sub-step is called *fusion*. It performs the merge between intermediate models in order to obtain the core requirements model. It resolves the redundancies and expresses clearly the links between the intermediate models.

3.3 Going from the RDL to the R2A

We have implemented the R2A platform that targets the integration of RE solutions into modern software development processes [7]. It has been designed as a result of our requirement analysis experiences in several industrial projects.

The work around the R2A platform has been initiated in a collaboration with THALES [8]. This project intended to evaluate the MDE approach to simulate requirements, and automate the generation of functional test objectives [9] from requirements.

The R2A platform has then been extended for modeling new telecom services in FRANCE TELECOM. This experiment aimed at simulating requirements, producing an analysis model and system test cases. More details are provided on the R2A website [10].

We have implemented a mechanism that follows the principles of the general process previously exposed (section 3.2). It is thus an instance of the generic process (Figure 2) in the R2A context that illustrates the general process.

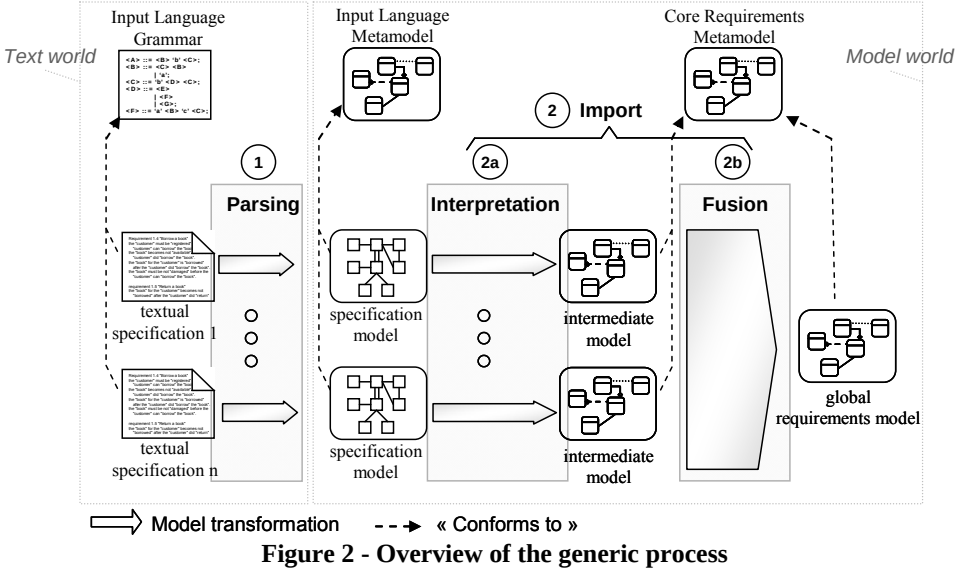

Figure 2 - Overview of the generic process

The Requirement Description Language (RDL) is a constrained natural language that has been designed as surface language for the R2A metamodel. Thus, all specification expressed with the RDL can be captured as a model instance of R2A. This language is also well adapted to the industrial contexts where we improve the platform. The sentences in Figure 1 are RDL expressions.

In the following, we focus on the parts of the R2A platform which are relevant for describing and illustrating the mechanism:

- Section 4 presents the metamodels for RDL and R2A and the parsing step.
- Section 5 presents the import step of the R2A platform. It describes the two transformations which perform the interpretation and fusion sub-steps.

4 The RDL Input Language and the R2A Output Metamodel

4.1 The RDL/RDM

The RDL is a pseudo-natural language (i.e. a constrained natural language). It has been designed to avoid writing ambiguous, incomplete and incorrect sentences. Its syntax allows writing readable sentences that look like natural language (although it is not intended to address any natural language analysis issues).

A RDL specification consists of a set of requirement sentences in which domain specific words are contained in quoted strings. These domain specific words include: the actors and the business concepts, the actions that actors can activate, the observable properties of business concepts or actors and the values of these observable properties. The textual specifications (a) and (b) in Figure 1 are two RDL expressions.

The RDM is a metamodel semantically equivalent to a RDL grammar. It describes the concepts of the RDL abstract syntax tree. A part of the RDM is presented in Figure 3. Requirements can be conditional or temporal. Temporal requirements (*RequirementTemporal*) are either synchronous (*RequirementBefore*, *RequirementAfter*) or asynchronous. They allow defining a variety of requirements by combining the different kinds of observable properties and service activations.

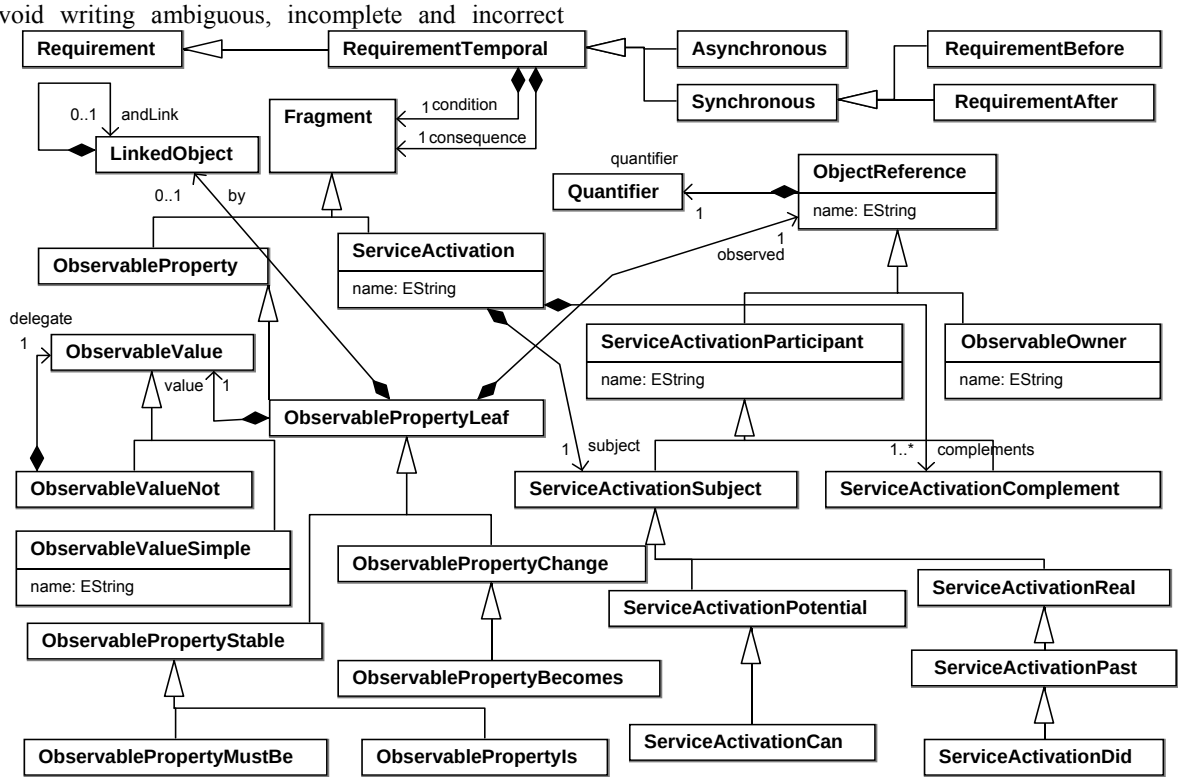

Figure 3 - The RDM metamodel

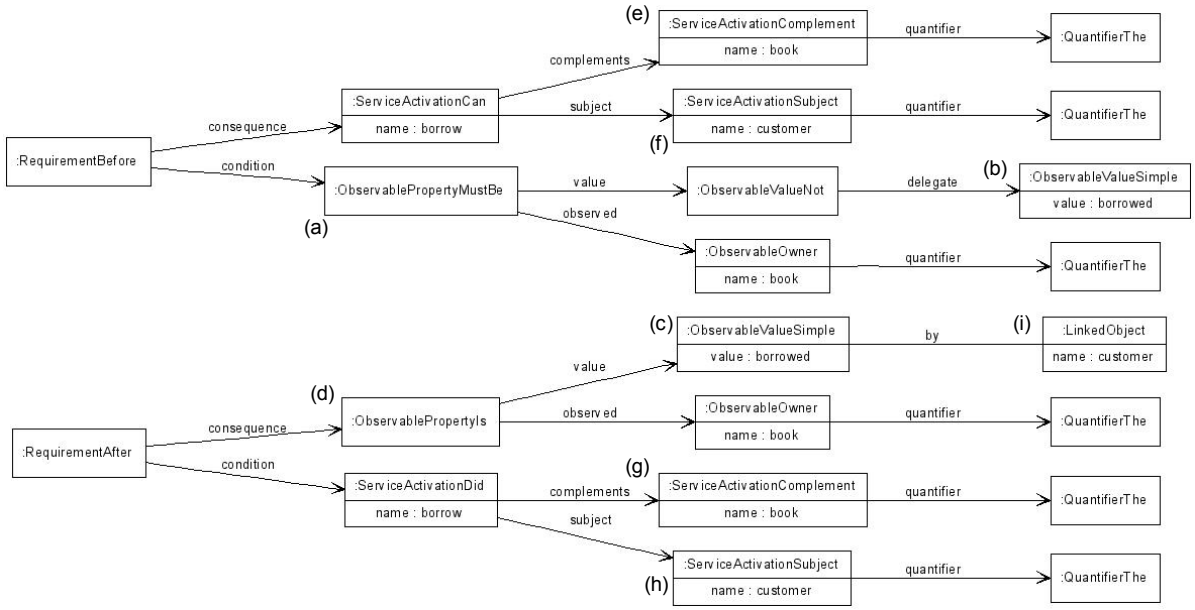

Figure 4 – A RDM model corresponding to the requirements (1) and (2) of the Figure 1

A requirement is composed of several propositions called *Fragment*. A fragment can be either a *ServiceActivation* or an *ObservableProperty*. Service activation corresponds to an action triggered by an actor of the system. It contains a *subject* (e.g. the service activator), a verb (determined by the type of the ServiceActivation) and a *complement* which describe the business concept or the actor impacted by the action (e.g. the service parameters).

The services are categorized according to their occurrence in time. For instance, *ServiceActivationCan* (of kind *ServiceActivationPotential*) represents a service whose activation depends on a specified condition (such as an observation on a system property). *ServiceActivationDid* (of kind *ServiceActivationReal*) represents a past service activation that can raise some changes in the system state (typically, the update of some system properties). The model depicted in Figure 4 contains an instance of both of these *ServiceActivation* types.

*ObservableProperty*s describe either constraints or assignments of actor or business concept states. For simplicity sake, we only consider here simple observable properties (*ObservablePropertyLeaf*). An observable property associates an *observed* property (*ObjectReference*) with a *value* (*ObservableValue*). Observable properties can be used to define conditions for service activations (constraints) as well as effects of these activations (assignments).

A constraint is expressed with an *ObservablePropertyStable* (*ObservablePropertyIs*, *ObservablePropertyMustBe*) while an assignment is represented by an *ObservablePropertyChange* (*ObservablePropertyBecomes*,*ObservablePropertyChanges*, etc.).

In order to bring RDL texts in the model world, we have used a dedicated MDE tool called *Sintaks* [11]. The resulting model is an instance of the RDM metamodel. It is an abstract syntax tree which corresponds to the semantics of RDL texts. The RDM model shown in Figure 4 is the result of the parsing step for requirements (1) and (2) of Figure 1. This model contains two instances of temporal requirements: *RequirementBefore* and *RequirementAfter*.

Sintaks is a model-based generic tool that can build a model representation from a text (parsing), but also pretty print a text from a given model. *Sintaks* has been built with a model-centric approach: the bridge between an abstract syntax (e.g. a metamodel) and a concrete syntax (e.g. a grammar) is specified by means of a dedicated *Sintaks* model. This model specifies the relationships between the metaclasses of a metamodel and the representation of their instances in textual files. Such a *Sintaks* file has been defined for the RDM metamodel. This file defines the concrete textual syntax of RDM models.

4.2 The R2A

In requirement specifications that we reviewed in the industrial contexts of THALES and FRANCE TELECOM, the description of system actions generally included conditions of activation which specify when the action is applicable. The description

also detailed the effects on the system and the environment if the action is triggered.

Based on this survey, we have defined the R2A metamodel. It captures a state-based formalism where system actions are described as use cases. The activation conditions of use cases and their effects are described by pre- and post-conditions (called *contracts* as for the design-by-contracts approach). Figure 5 gives a textual representation of a R2A model obtained with the platform from LMS requirements (2, 6, 7, 8).

Use Case *Borrow (c: Customer, b: Book)*
Pre*: registered(c)* ***and*** *registered(b)* ***and not*** *damaged(b)*
and not *borrowed (b)*
Post*: borrowed (b)*

Figure 5 - A R2A model in a textual form

The R2A is structured in two main parts: (i) a description of use cases with their contracts, called *FunctionalModel* and (ii) a description of entities manipulated by the use case contracts, termed *AnalysisModel*. We detail them in this section.

a The Functional View

As shown by Figure 5, the R2A allows specifying: when the use case is applicable (precondition), and what are the modifications on the system properties after activation of the use case (post-condition). In other words, the precondition guards the execution of the use case while the post condition specifies its impact on the system if triggered.

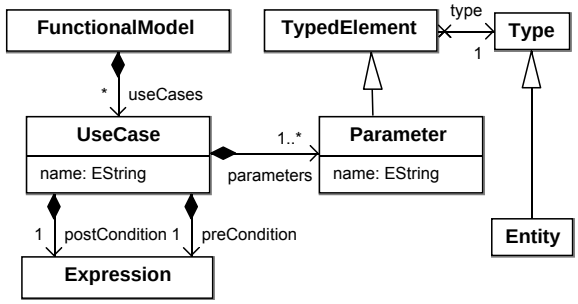

Figure 6 - Functional view of the R2A metamodel

Figure 6 details the *FunctionalModel* part of the metamodel. A use case is represented as an instance of the class USECASE and has exactly one pre-condition and one post-condition. It contains a set of formal parameters (PARAMETER class). A formal parameter type can be either an actor or a business concept (ex: *Customer* and *Book*) which are both represented by the class ENTITY.

Contracts, represented by EXPRESSION class, are first order logical expressions. We do not present in details the expression part of the metamodel due to space limitation. An expression is a combination of logical operators, boolean comparisons and atomic propositions. Logical operators can be conjunction (*and*), disjunction (*or*), negation (*not*), implication or quantifiers (*exists* and *forall*). Atomic propositions are either boolean constants (true and false), or property values of actors and business concepts. The notion of property value is defined below.

b The Analysis View

Use cases involve business concepts and actors (the formal parameters) as well as predicates on a set of object called property values (in the contracts). For example, `Customer` is an actor; `Book` is a business concept; `registered` is a property while `registered(b)` is a property value. Actors, business concepts and their properties are described by the *AnalysisModel* part of the metamodel, presented in Figure 7.

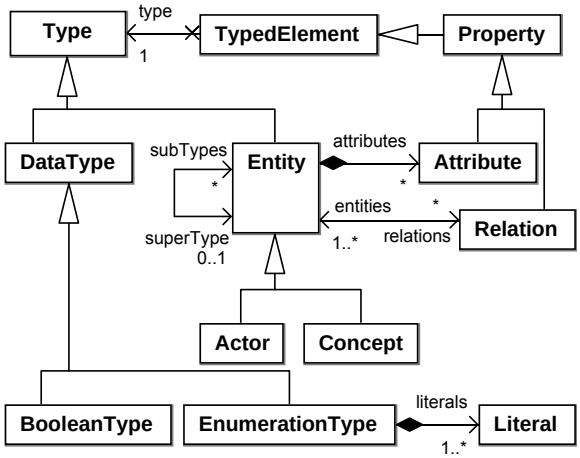

Figure 7 - Analysis view of the metamodel

An actor or a business concept is represented by an ENTITY. It is linked to a set of PROPERTY which can be a RELATION between several entities or a value attached to one entity (ATTRIBUTE). An attribute has a primitive data type which is either a BOOLEAN or an ENUMERATION. An enumeration is a finite set of literal values (an integer or an interval is represented by a literal value). Complex data types are not supported since we consider they should not be present in a requirements model.

Instances of attributes and relations are property values of the system. A system state is a set of entity instances and property values. It corresponds to a configuration of the system at a given moment.

5 Interpretation and Fusion Transformations: RDM to R2A

Here we show how RDM models are transformed to feed the global R2A requirements model (step 2). We illustrate the interpretation step with LMS requirements (1, 2) and the fusion step by merging two

intermediate models obtained from the requirements (1, 2) and (6, 7, 8) (these requirements together describe the "borrow" action).

5.1 Interpreting RDM Models as R2A Models

Interpreting a RDM model as a R2A model supposes generating both analysis and functional parts of the R2A model. The interpretation step is performed by a model transformation. In the context of the R2A platform, we implemented this step with a dedicated language called IRL (Interpretation Rule Language).

An IRL program is a set of *Interpretation rules* (IR) which consist of a *pattern* and a *production*. A pattern defines the set of input model elements on which the associated rule matches. This set is defined by the type (*pattern_type*) of its elements (a metaclass) and a condition (*pattern_cond*) that each matched element must satisfy. The type is a first filter in order to select the matched elements.

The production describes the model elements created and the way they will be linked if the rule is executed. The produced elements are instances of the output metamodel. The resulting model is then a set of model fragments which are not interlinked. An IR specification uses the following template:

IR*number* (*elt* : *pattern_type* (| *pattern_cond*)?): *informal_production_description*.

An IR is identified by a number. Its pattern is outlined by parenthesis where the matched element has a type (a RDM metaclass in the R2A context) and the pattern condition is optional. The production is described informally but contains OCL-like navigation expressions inside the RDM metamodel.

In our case, 17 IRs have been defined: 7 for producing elements relative to the analysis model and 10 for the functional model. The list below gives the interpretation rules which are necessary for producing an R2A analysis model from the RDM model presented in Figure 4.

IR1 (o: *ObservablePropertyLeaf*): creation of a CONCEPT which contains an ATTRIBUTE. The *name* of the concept is o.observed.name. The *name* of the attribute is o.value.value (or o.value.delegate.value if o.value is a OBSERVABLEVALUENOT).

IR2 (o: ObservableValueSimple | o.by is not set): creation of an *Attribute* with a type BOOLEANTYPE.

IR3 (o: ObservableValueSimple | o.by is set): creation of a RELATION. Its type is a BOOLEANTYPE and it is linked to a CONCEPT which the *name* is o.by.name.

IR4 (s: ServiceActivationSubject): creation of an ACTOR which the *name* is s.name.

IR5 (s: ServiceActivationComplement): creation of a CONCEPT which the *name* is s.name.

An IR execution computes a relation between two model fragments: the set of classes and relation instances from the input model which match the pattern condition or are navigated by the production; the set of class and relation instances from the output model which are created by the production.

Figure 8 gives two examples of IR executions on the RDM model shown in Figure 4. The first one (on the left) illustrates the execution of the rule IR1. The matched element is the OBSERVABLEPROPERTYIS instance (d) in Figure 4. A CONCEPT and an ATTRIBUTE are then created and linked by an *attributes* reference instance. The first one takes as *name* the *value* of the OBSERVABLEVALUESIMPLE instance. The second one takes as *name* the *name* of the OBSERVABLEOWNER instance.

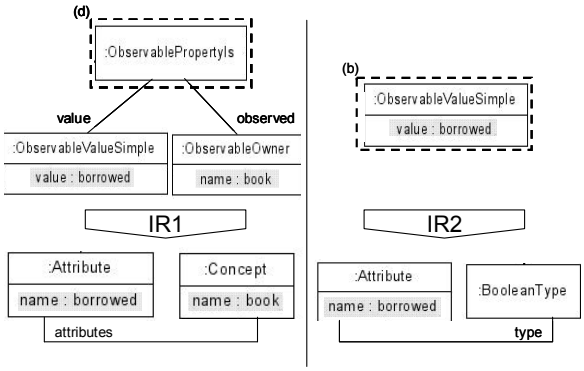

Figure 8 - Illustration of IR executions

The second example depicts the application of the rule IR2 on the element (b) in the RDM model (Figure 4). In that case, the matched element satisfies the pattern condition (no element connected by a *by* relation instance). In contrary, element (c) does not satisfy the pattern condition but it does match the pattern type.

Figure 9 presents the intermediate R2A model that we obtain by completely executing these five IRs on the RDM model (Figure 4). The model elements are grouped according to the rule execution which has produced them. Thus, we find the two model fragments whose the production has been described in Figure 8. These model fragments are labeled by the name of the rule which created them and the matched element in Figure 4. For example, "IR1→a" means "created by IR1 by matching element (a)".

In a complex transformation process, it is important to notice that a model does not continuously conform to its metamodel during the execution of a transformation. We can notice that the intermediate model depicted in Figure 9 does not conform to its metamodel. Indeed, the R2A metamodel declares that a TYPEDELEMENT instance is inevitably linked to a TYPE instance in a valid R2A model (the cardinality is one).

However, two ATTRIBUTE instances out of three have no type in the intermediate R2A model.

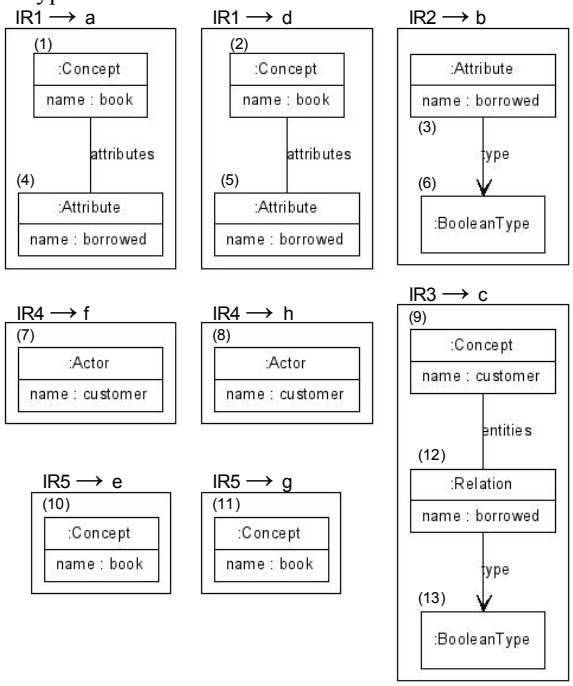

Figure 9 – The analysis part of the intermediate model obtained from requirements (1) and (2)

This non conformance reflects the redundancy of the RDL text. Moreover, producing redundancy during the interpretation step is not a problem since it will be resolved by the fusion sub step. In fact, step 2a and 2b are the two parts of a global model transformation for translating a requirement specification from one language to another. We present in the next section the fusion step.

5.2 The Fusion Semantics of the R2A Metamodel

The fusion step aims to merge several intermediate models in order to obtain the global requirements model. It is performed by another model transformation where the input and the output metamodels are the R2A metamodel. It has been implemented using the Kermeta language [12, 13] which is an open source meta-modeling environment, developed by the Triskell team at IRISA.

In practice, the fusion transformation is asymmetric in the sense that it distinguishes one particular model as being the core requirements model in which all the intermediate models have to be merged. The first time this fusion is performed, one of the intermediate models is chosen arbitrarily as the core. The fusion consists in adding all elements of the intermediate models in the core and in merging all similar elements (redundancies) in order to obtain a correct and compact representation of the requirements model. Some rules, called implicit rules, aim to normalize the resulting model. We give an example of implicit rule in the following.

The fusion transformation is defined with a set of dedicated rules called *fusion rules* (FR). As introduced in section 2.3, two kinds of FRs can be defined: equivalence and implicit rules. An equivalence rule targets the resolution of redundancies. Equivalence rules are used to define *equivalence range*. An equivalence range is a set of elements in the intermediate model which must be represented by only one element in the global requirements model. Implicit rules define fusion treatments which produce specific elements in the global requirements model.

In this paper, FRs are provided as first order logic expressions (except for implicit rules which are generally more complex) extended with specific notations for manipulating models and two specific operators: the *equivalence declaration operator* ('≡') and the *equivalence resolution operator* (':≡'). The first one is used for defining equivalence range (a ≡ b and b ≡ c means that {a, b, c} is an equivalence range). The second one describes which element will be chosen in an equivalence range for representing the concept in the global requirements model.

The fusion transformation of the R2A platform is composed of 30 rules: 10 rules for merging elements relative to the analysis model and 20 for the functional model. The FRs below are an excerpt of this FR set. They manipulate instances of the R2A metamodel presented in section 5. Rules FR1 to FR5 are equivalence rules specific to the analysis model. Rule FR6 is an implicit rule specific to the functional model.

FR1: $\forall$ a_1, a_2 : ATTRIBUTE $\wedge$ $a_1.name = a_2.name \wedge a_1.attributes^{-1} = a_2.attributes^{-1} \Rightarrow a_1 \equiv a_2$.

FR2: $\forall$ r_1, r_2 : RELATION $\wedge$ $r_1.name = r_2.name \Rightarrow r_1 \equiv r_2$.

FR3: $\forall$ r : RELATION $\wedge$ $\forall$ a : ATTRIBUTE, $a.name = r.name \Rightarrow r :\equiv a$.

FR4: $\forall$ c_1, c_2 : CONCEPT $\wedge$ $c_1.name = c_2.name \Rightarrow$ ((c_1 is an ACTOR $\Rightarrow c_1 :\equiv c_2$) $\vee$ $c_2 :\equiv c_1$)

FR5: $\forall$ b_1, b_2 : BOOLEANTYPE, $b_1.type^{-1} = b_2.type^{-1} \Rightarrow b_1 \equiv b_2$

FR6: (uc_1, uc_2 : USECASE | $uc_1 \equiv uc_2$): For n preconditions, creation of n-1 ANDEXPRESSION instances. Creation of links between preconditions.

Figure 10 shows a global requirements model obtained from two intermediate models produced from requirements (1, 2) (an excerpt is given in Figure 9) on the one hand and from requirements (6, 7, 8) on the other. The gray elements have been produced by the FRs (1-5) above. Figure 10 gives an overall

specification of the "borrow" action of the LMS since this action is described by requirements (1, 2, 6, 7, 8).

Fusion rule 3 (FR3) defines that RELATION and ATTRIBUTE instances which have the same name in the model are equivalent. This rule is applied on the elements (3, 4, 5, 12) in the intermediate model presented in Figure 9. As a result, only one instance of the RELATION R2A metaclass with the name "borrowed" appears in the global requirements model provided by Figure 10. We can notice that an equivalence rule execution reduces the number of elements which will be present in the model.

When equivalent use cases are merged, the relationship precondition in the analysis part of the R2A metamodel is likely to be instantiated more than one time for the resulting use case. A use case with several pre-condition is semantically correct: the pre-condition is conjunction of them. However, it is not correctly expressed w.r.t. the metamodel. The pre-condition must be a logical tree (as in the Figure 10). Implicit rule FR6 aims to normalize this kind of situation. It creates the conjunction links between the preconditions of a particular use case. Its condition of activation (outlined in parenthesis) specifies that it is applied on each pair of USECASE instances that have been declared as equivalent. A conjunction is represented by an instance of the ANDEXPRESSION R2A metaclass (this metaclass is included in the expression part of the R2A metamodel). A similar rule exists in the R2A platform for the postconditions.

In Figure 10, a BOOLEANPROPERTY instance represents an instance of a property for one particular parameter of a use case (indeed, it is possible to have two instances of the same entity involved as parameters of one use case). While we specify the link to the property with the attribute *name*, another solution could be to have a link to the targeted property instead.

6 Related Work

The issue of managing several partial requirement specifications or viewpoints has received a lot of attention in the RE domain. In [4], Easterbrook defines a framework for managing inconsistencies between viewpoints and a formalism for expressing these inconsistencies. This framework aims at revealing the assumptions made by the stakeholders during the requirement elicitation stage. An inconsistency is detected if two viewpoints define in a different way a same part of the global specification. The inconsistency is resolved when the authors of the two viewpoints agree on a common description, taking into account the concerns of each author in their respective viewpoints.

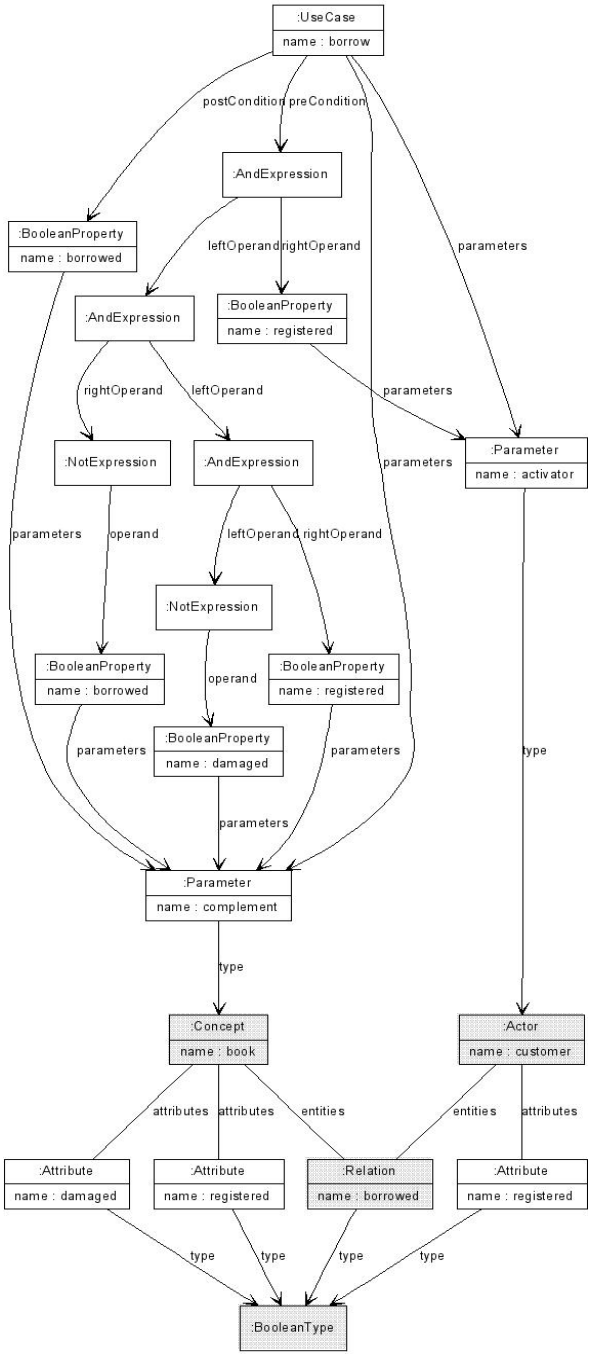

Figure 10 - A core R2A model generated by the platform for LMS requirements (1, 2, 6, 7, 8)

In our approach, each viewpoint is initially considered as correct. Inconsistencies are detected by confronting stakeholders with the model obtained by merging their viewpoints. Both approaches are in fact complementary: The Easterbrook's framework allows to detect conceptual disagreements. Once they are

resolved, our approach can merge models in one language, depending of the targeted analysis method.

The viewpoint framework allows checking a wide range of syntactical inconsistencies in terms of relations between abstract syntax of viewpoints languages (notations in [4]). However, one significant limitation is that semantic inconsistencies such as logical contradictions between viewpoints can not be detected. As stated by the authors [4], semantic consistency checking necessitates a formal model representing the semantics of the overall requirements specification. This model is obtained by merging the viewpoints together. Our approach allows obtaining such a global semantic model. It is thus possible to implement mechanisms for verifying the semantic correctness of the global specification.

Several works present viewpoints merge mechanism (also called amalgamation or composition) [14, 15]. In [14], the authors propose an amalgamation mechanism for producing a global Z specification from several Z specifications. The relations between elements of different viewpoints are described with functional invariants. By using the Z language, they are able to provide proof obligation for verifying the amalgamation. Although our approach is less formal, it is more general since input languages as well as the core requirements metamodel are flexible.

The composition proposed in [15] is based on a core formalism called "semantic domain" and each input language comes with a function for assigning it a semantics in the semantic domain (as the interpretation step). The result of such a function is a set of "specificands" (an intermediate model for us) expressed in the semantic domain. The authors focus in the paper on the description of a semantic domain which can take into account a wide range of input languages. This semantic domain is the set of standard models of one-sorted first-order predicate logic with equality, enhanced with the notion of event, sequences, time intervals and markers. In that context, the composition of partial specifications is the conjunction of the corresponding sets of specificands.

The formalism of the R2A platform (the R2A metamodel) is based on first-order predicate logic where actions are the events. It is less expressive than the semantic domain described in [15]. However, it is sufficient for capturing the requirements of the projects we studied in FRANCE TELECOM and THALES. While the approach proposed in [15] is closed to our work, the authors do not focus on how to implement the composition of viewpoints. Our process decomposes the composition treatment in order to make it flexible. Furthermore, it has been designed for capturing requirements expressed with different textual syntaxes. The process can be used also for non-textual input languages (in that case, one has to skip the parsing step).

MDE is an approach that can help in the task of keeping generated software development artifacts up to date in response to requirement changes. It advocates the use of tools for generating features (parsers, editors, etc.) which are dedicated to a particular metamodel. In [16], the authors advocate, like us, the use of the meta-modeling approach, so that the input languages can be customized to the user needs. Our process is completely based on a model-driven approach. Sintaks is a good example of MDE tools. It allows defining easily a concrete syntax and a parser for a metamodel. Similar tools are also available for graphic languages.

The general process proposed in this paper is a basis for performing requirement analysis activities, such as detecting inconsistencies. It is a necessary step in addressing the issue of integrating requirements in modern development processes. Requirements engineering and software development remain indeed two very distinct tasks in the current software developments. Sommerville gives four reasons [7] for which this lack of continuity between requirements and software development becomes a crucial issue.

Recent tools for requirements analysis tend to bridge the gap between requirements engineering and software development by defining, like us, a core model that represents the captured requirements. Several inputs are used to populate the core model, like constrained natural languages and graphical languages (UML etc.). The core model is then transformed into one or several output models suitable for property-checking tools. For instance, [17] presents an integrated tool suite called SPIDER. It provides analysis of temporal behavioral properties on UML models with the SPIN model-checker. The tool is used to analyze systems whose implementation follows the MDA principles [6] (separation business and application logic from the underlying technological platform). The properties are expressed in a constrained natural language whose accepted sentences match well the Dwyer temporal logic patterns [18].

In the same way, we define a constrained language (RDL) for capturing business and functional concerns. The SPIDER input languages are UML class and state diagrams which are parts of the analysis model in a current development process. Instead of assuming that requirements are directly expressed with UML models, the R2A platform captures information from textual requirement specifications. Furthermore, SPIDER and other tools are not flexible w.r.t. their input language.

7 Conclusion

In a context where requirements are quickly changing and are described by several stakeholders, it is crucial to have a global requirements model to automatically compare the ways the stakeholders think the software-to-be or to analyze the impact of requirements changes. In this way, appropriate and rapid decisions can be taken quickly for avoiding common software development pitfalls.

In this paper, we have presented a general process which aims to facilitate the production of a global requirements model from multiple partial requirements specifications. This model is a semantic merge of the input requirements specifications where implicit links between them are clearly expressed. The process is based on a model-driven approach and has been designed to be highly flexible.

We have described an implementation of this mechanism in the R2A platform. It takes as inputs a set of RDL texts and produces an R2A model. The RDL allows expressing a wide range of functional requirements. The R2A metamodel was designed with two industrial partners for incrementally defining requirements by simulation, generating test objectives and analysis models.

The presented mechanism is an incomplete solution for involving more deeply RE activities into modern software development process. But it is at least one step towards this purpose. Beside a formalization of the process, our future work will address activities both upstream and downstream in the software development process. Upstream, we will focus on the detection and the localization of requirements inconsistencies between requirement specifications written in several input languages. Downstream, we will be interested in mechanisms for propagating easily requirements changes to software artifacts.

8 References

1. Nuseibeh, B. and S. Easterbrook. *Requirements Engineering: A Roadmap*. in *The Future of Software Engineering (joint event of ICSE'00)*. 2000. Limerick, Ireland.
2. The_Sandish_Group, *Chaos Standish Group Internal Report*. 1995, The Standish Group.
3. META_Group, *Research on Requirements Realization and Relevance*. 2003.
4. Easterbrook, S. and B. Nuseibeh, *Using Viewpoints for inconsistency management.* Software Engineering Journal, 1996. **11**(1): p. 31-43.
5. Schmidt, D.C., *Model-Driven Engineering.* IEEE Computer, 2006. **39**(2): p. 25 - 31.
6. OMG. *MDA*. 2003 [cited 2005; Available from: http://www.omg.org/mda/.
7. Sommerville, I., *Integrated Requirements Engineering: A Tutorial.* IEEE Software, 2005. **22**(1): p. 16-23.
8. Lugato, D., et al. *Automated Functionnal Test Case Synthesis from THALES industrial Requirements*. in *RTAS'04*. 2004. Toronto, Canada.
9. Nebut, C., et al., *Automatic Test Generation: A Use Case Driven Approach.* IEEE Transactions on Software Engineering, 2006.
10. Saeed, W.A., et al. *Model Driven Requirements Engineering*. 2006 [cited; Available from: http://www.irisa.fr/triskell/Softwares/protos/mdre/.
11. Muller, P.-A., et al. *Model-Driven Analysis and Synthesis of Concrete Syntax*. in *MoDELS'06*. 2006. Genova, Italy.
12. Kermeta. *The KerMeta Project Home Page*. 2005 [cited 2005; Available from: http://www.kermeta.org.
13. Muller, P.-A., F. Fleurey, and J.-M. Jézéquel. *Weaving executability into object-oriented meta-languages*. in *MoDELS'05*. 2005. Montego Bay, Jamaica: LNCS.
14. Ainsworth, M., et al., *Viewpoint Specification and Z.* Information and Software Technology, 1994. **36**(1): p. 43-51.
15. Zave, P. and M. Jackson, *Conjunction as Composition.* Transaction on Software Engineering and Methodology, 1993. **2**(4): p. 379-411.
16. Nissen, H.W., et al., *Managing Multiple Requirements Perspectives with Metamodels.* IEEE Software, 1996. **13**(2): p. 37-48.
17. Konrad, S. and B.H.C. Cheng. *Automated Analysis of Natural Language Properties for UML Models*. in *MoDeVa 05 workshop in conjunction with MoDELS'05*. 2005. Montego Bay, Jamaica.
18. Dwyer, M.B., G.S. Avrunin, and J.C.i.I. Corbett. *Patterns in Property Specifications for Finite-State Verification*. in *ICSE'99 (Int. Conf. Software Engineering)*. 1999. Los Angeles, CA, USA.